NC STATE UNIVERSITY

06: Scheduling and Dispatching, Response Time Analysis and OS Wish List

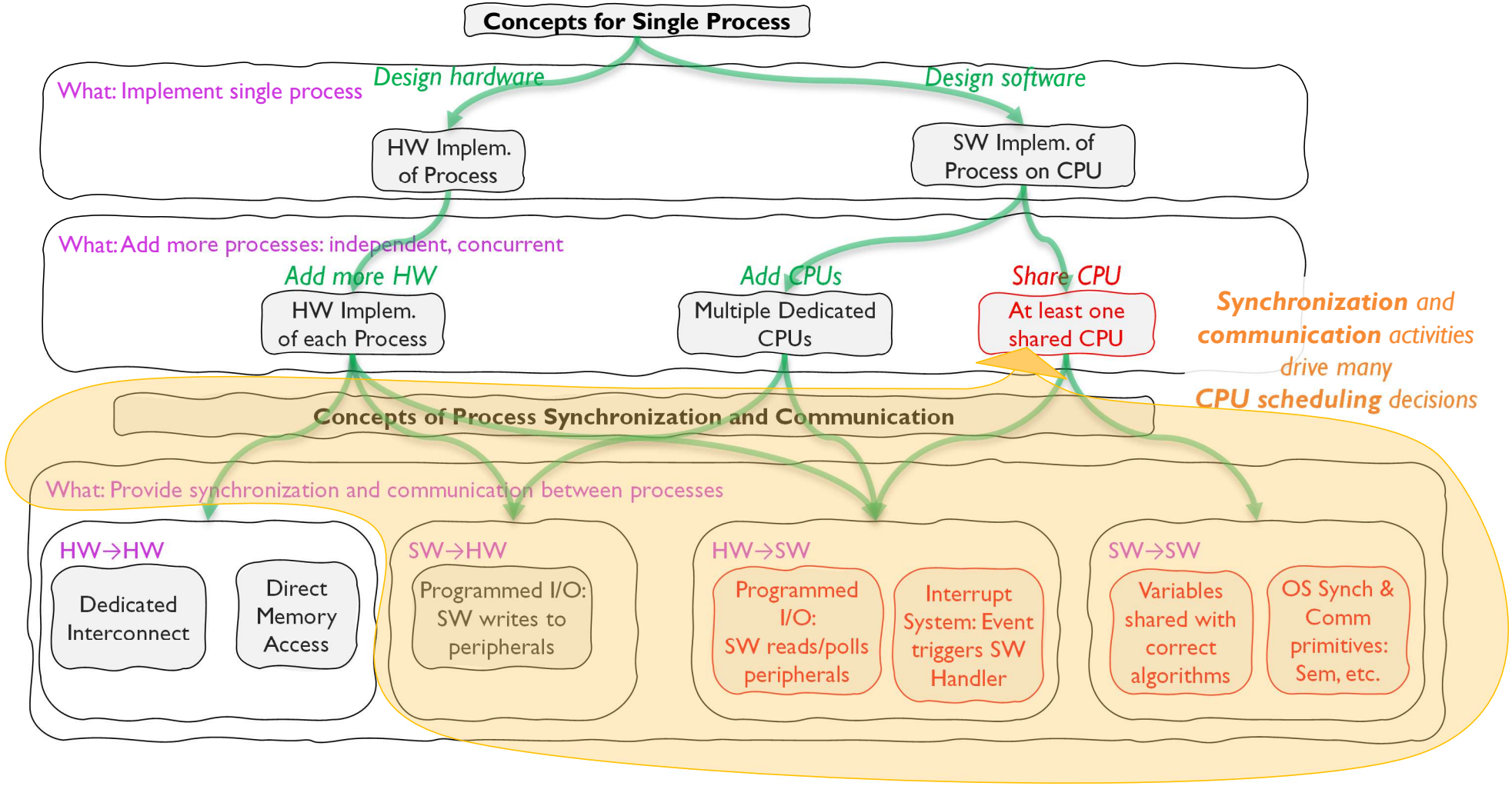
9/8/2025

NC STATE UNIVERSITY

Overview

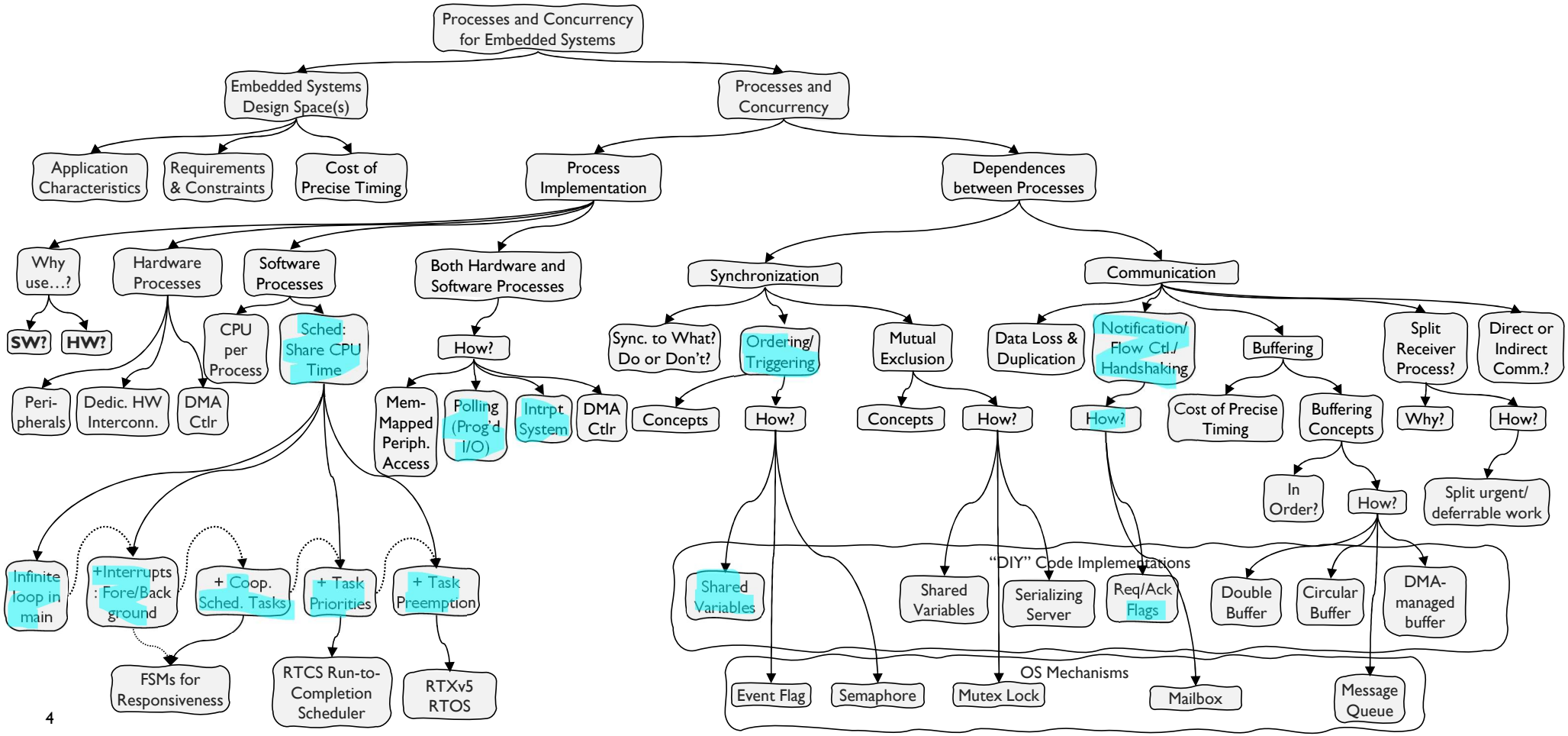
- Where are we?
- Examining the Processing Chain
  - Scheduling and Dispatching: Where are they done?
- Response Time Analysis
- OS Wish List

SW Processes: CPU Scheduling, Synchronization & Communication



NC STATE UNIVERSITY

Extended Topic Map: Class 06



Software Processes and Arm CPU Modes

- Arm CPU may run a SW process in…
  - Thread Mode
  - Handler Mode
- Main differences: *These don't matter yet*
  - Which stack pointers (SP) are available
    - Main SP, Process SP
  - Which access privilege levels are available
    - Unprivileged level prevents using certain instructions and accessing certain peripherals, control registers and memory regions
    - Privileged level has no restrictions
- Transitions
  - Thread mode -> handler mode: When starting to respond to an interrupt or exception request
  - Handler mode -> thread mode: after finishing handling last nested interrupt or exception request

NC STATE UNIVERSITY

Processing Chain: Schedule and Dispatch Stages

NC STATE UNIVERSITY

Processing Chain Refinement

- Stages
  - Detect Event: Sample and quantize signal, analyze and decide if event happened
  - Schedule process: Decide what processing to do next (e.g. do process work)
  - Dispatch process: Start it running (or resume it)
  - Do: Perform processing work to handle event

Dispatching a Software Process

Comes after scheduling, but let's get it out of the way early

- Dispatch = cause SW process to start/resume executing
- Different methods available
  - Implicit: next instruction in code is part of the process
  - Subroutine call to with process' root (overall) function
  - Interrupt Controller forces CPU to execute ISR containing process root function
  - Interrupt Controller forces CPU to execute handler, which then uses OS to switch contexts and resume running the process.

NC STATE UNIVERSITY

Scheduling & Dispatching: Decide what to do next, and start it

- Behavior depends on two decisions
- 1. Is this process **allowed to run**?
  - Yes: Dispatch and run it
  - No: 2. What kind of event detection **test**?
    - Non-blocking: Advance to detect stage for next process (via scheduler or program structure) and continue
    - Blocking: Repeat software starting with detect (analyze and decide) by looping back to it

Detect Event				
Analyze	Decide	Schedule	Dispatch	Do Work

System Timelines for Non-Blocking vs. Blocking Detection

- Resulting system timelines
  - Non-blocking detection: round-robin
  - Blocking detection: greedy

Processing Chain Variations 1: Where to Detect & Schedule?

```
main() {
   while (1) {
     // Detect Event for A
     ev_A_det = ….
     if (ev_A_det) {
          ev_A_det = 0;
          A_Work();
     }
     // Detect Event for B
     ev_B_det = ….
     if (ev_B_det) {
          ev_B_det = 0;
          B_Work();
}}
```

- Main Thread Loop: Scheduling loop in main thread
  - Main polling event detection code

```
A_Det_Sched_Work( ) {
   ev_det = …
   if (ev_det) A_Work();
}
B_Det_Sched_Work( ) {
   do { // blocking
     ev_det = …
   } while (!ev_det);
   B_Work();
}
main() {
   while (1) {
     A_Det_Sched_Work();
     B_Det_Sched_Work();
}}
```

- Process: in Do Work
  - May have built-in Detect/Schedule/Do
  - Schedule may be single non-blocking test or looping blocking test

```
ISR_A( ) {
   A_AllWork();
}

main() {
   while (1) {
     …
}}
```

- Interrupt System: Hardware
  - Peripheral detects, interrupt controller schedules & dispatches, interrupt handler (ISR) does all the work

Processing Chain Variations 2: Where to Detect & Schedule?

- Interrupt System & (Main or Process): Multiple locations
  - Combines interrupt approach with another
    - Allows splitting of work between ISR and thread for better responsiveness
    - Needs synchronization between processes
    - Results in foreground/background system
  - Operation
    - Peripheral detects, interrupt controller schedules, handler does some work and requests more processing (ev_A_det)
    - Main loop detects request, schedules process, process does requested processing work
      - Could instead use A_Det_Sched_FinishWork()

```
volatile int ev_A_det = 0;

ISR_A( ) {
   A_StartWork();
   ev_A_det = 1;
}
A_FinishWork( ) {
   …
}
B_Det_Sched_Work( ) {
   do {
      ev_det = …
   } while (!ev_det);
   B_Work();
}
```

```
main() {
   while (1) {
      if (ev_A_det > 0) {
           ev_A_det = 0;
           A_FinishWork();
      }
      B_Det_Sched_Work();
   }
}
```

NC STATE UNIVERSITY

Response Time Analysis

NC STATE UNIVERSITY

Starting Point for Response Time Analysis

- Response time = time between **event** and **completion of response processing**
- RTA for **which** process?
  - This process?
  - Other processes in the system? How does this process affect/disrupt timing for the other processes in the system?
- Managing variations in processing chain structure
  - Standardize to simplify timing analysis:
    - Assume detection (analyze, decide) and simple mini-scheduler (if) for process is performed in its Do Work stage
      - Still have outer scheduler deciding which process to run next
    - Process Do Work stage is short if event not detected, longer if event is detected
  - Can link multiple processing chains together if the processes synchronize with each other

Overview of Basic Approach to RTA

- Two parts
  - How much **processing time** is needed for **software process A**'s instructions?
  - **What else can run** between input event at t_{A_event} and response processing completion at t_{A_done}? **How long does it take**?
    - Includes other processes and the scheduler/operating system process(es)

- Breaking down **what else**
  - **How soon** does the scheduler start **running A**?
    - Is a **process already running** that will delay when the scheduler gets to run?
    - Does the scheduler have **other processes to run before** starting A?
  - Can anything **delay A after it has started**?
    - Can anything **preempt** A after it starts running?
    - Could A have to **wait for another process** for synchronization?

NC STATE UNIVERSITY

RTA Examples with Four Schedulers

- Workload
  - Processes: P1, P2, P3, P4
  - Triggering Events (or conditions): E1, E2, E3, E4
- V1. Fixed order round-robin
- V2. Interrupt detects event E1, Interrupt handler sets flag requesting a run of P1
- V3. Prioritize processes. P1 > P2 > P3 > P4
- V4. Move P1 work into ISR

NC STATE UNIVERSITY

Response Times for Schedulers

	Release Schedule R1	Release Schedule R2	Release Schedule R3
P1			
P2			
P3			
P4			

NC STATE UNIVERSITY

Observations

Observations

- Limitations of run-to-completion process model.
  - Thread duration vs. responsiveness
  - Thread preemption only by interrupts, complicating design
  - Shortening threads with finite state machines

- Sync/comm/sched/dispatch operations are often interdependent
  - If scheduler/OS can see all these operations, it can make better decisions and offer more features
  - Example: If process event test will block, then pause process execution and automatically switch in another process. Reclaims idle time.

NC STATE UNIVERSITY

Observations

- Scheduling model variations: Where are detect and schedule performed?
  - A. Main sched thread (Det, Sched) only: in main loop with polling detection
  - B. Interrupt only: peripheral detects, interrupt controller schedules, handler does
  - C. Main & Interrupt system:
    - Peripheral detects, interrupt controller schedules, handler does some work and requests more processing
    - Main loop detects by polling request, scheduling process, process does requested work
  - D. Do Work/Handler
    - Do Work/Handlerportion of process may also contain Get/Detect/Schedule/Do built in, where Schedule step may be single non-blocking test or looping blocking test

OS Wish List

Wish List - Better control of responsiveness

- Methods to provide ("allocate") responsiveness to processes (or parts within) as needed
  - Don't waste responsiveness on processes which don't need as much
  - Reduce vulnerability of responsiveness for urgent processing by isolating it from less urgent processing
- Improvements
  - Improve task execution order (not round-robin A B C D A B C D)
  - Add process priorities, use to drive scheduling.
    - Static priority?
    - Dynamic? Based on slack time?
    - Both?
  - Improve run-to-completion processes (non-preemptive) with yield and resume features: Finite state machines (FSMs), other methods. Cooperative multitasking
  - Provide preemption of processes by higher priority processes

Generalize/standardize code structure for modular code

- Standardize data format for scheduler
  - Essential data: process is ready (has permission to run since event was detected). Count to 1 or higher?
- Provide protected interface for scheduler data. E.g. request another run.
- Support scheduling decisions more locations: scheduler code, user code

Features to simplify programming

- Support time-based process scheduling (e.g. with periodic timer tick)
  - Run this process every N ticks, etc.
- Features for synchronization between processes:
  - Signaling event has occurred, counting pending un-serviced events. E.g. for triggering processing: ISR->thread, etc.
  - Protecting critical sections with mutually exclusive execution.
- Features for communication between processes
  - Send a message: data and provide sync support for receiver (and sender too!)
  - Send a message, allowing multiple pending messages (FIFO/queue)

NC STATE UNIVERSITY

Features to reclaim idle time

- Take advantage of OS knowledge of system state
  - Switch processes when blocking

- Leverage preemption